Fast Tridiagonal Solvers on GPU

Yao Zhang
John Owens
UC Davis

Jonathan Cohen
NVIDIA

GPU Technology Conference 2009

Outline

- Introduction
- Algorithms
  - Design algorithms for GPU architecture
- Performance
  - Bottleneck-based vs. Component-based performance model
- Summary

What is a tridiagonal system?

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{pmatrix}$$

What is it used for?

- Scientific and engineering computing
  - Alternating direction implicit (ADI) methods
  - Numerical ocean models
  - Semi-coarsening for multi-grid solvers
  - Spectral Poisson Solvers
  - Cubic Spline Approximation
- Video games and computer-animated films
  - Depth of field blurs
  - Fluid simulation

A Classic Serial Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} 1 & c_1' & & & \\ 0 & 1 & c_2' & & \\ & 0 & 1 & c_3' & \\ & & 0 & 1 & c_4' \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d_1' \\ d_2' \\ d_3' \\ d_4' \\ d_5' \end{pmatrix}$$

Phase 2: Backward Substitution

Parallel Algorithms

- Coarse-grained algorithms (multi-core CPU)
  - Two-way Gaussian elimination
  - Sub-structuring method

A set of equations mapped to one thread

- Fine-grained algorithms (many-core GPU)
  - Cyclic Reduction (CR)
  - Parallel Cyclic Reduction (PCR)
  - Recursive Doubling (RD)
  - Hybrid CR-PCR algorithm

A single equation mapped to one thread

A little history

- Parallel tridiagonal solvers since 1960s:
  - Vector machines: Illiac IV, CDC STAR-100, and Cray-1
  - Message passing architectures: Intel iPSC, and Cray T3E
  - And GPU as well!




Two Applications on GPU

Depth of field blur, Michael Kass et al.

OpenGL and Shader language

Cyclic reduction

2006

Shallow water simulation

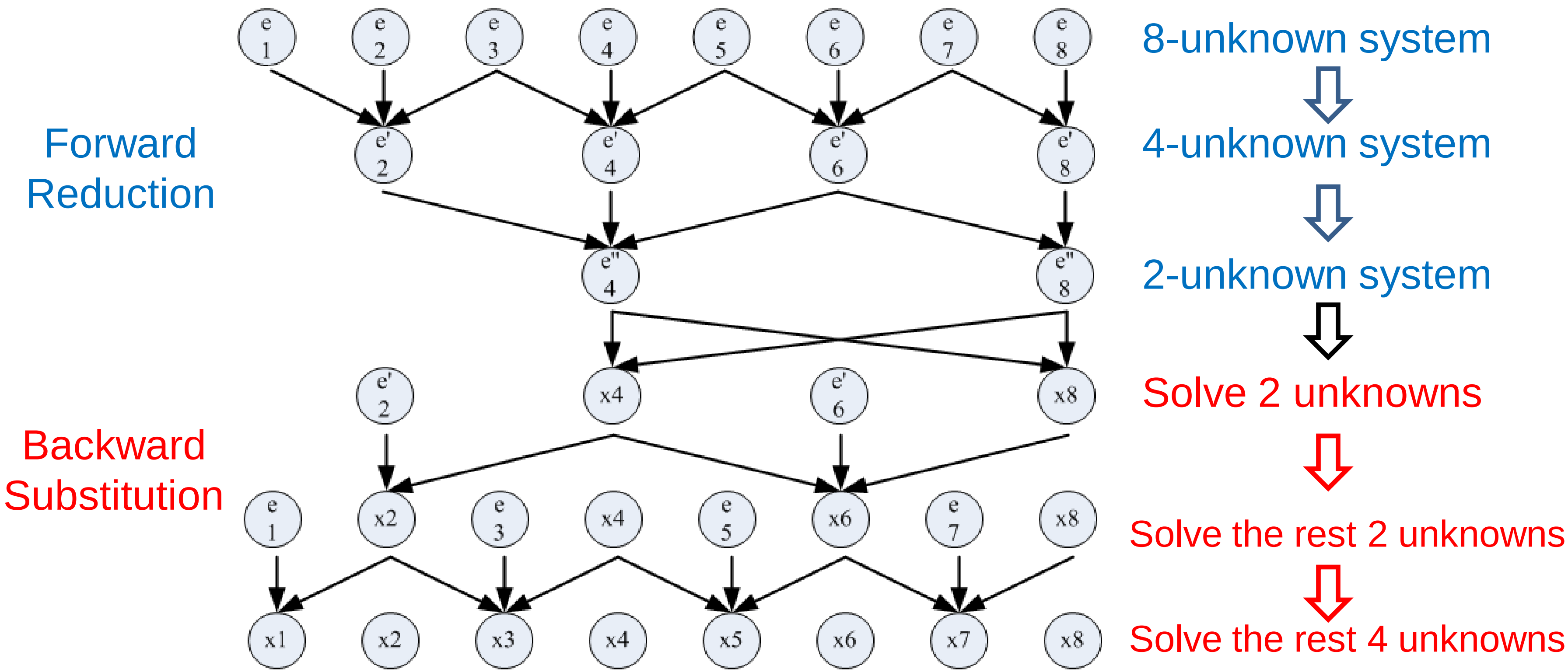
CUDA

Cyclic reduction

2007

Cyclic Reduction (CR)

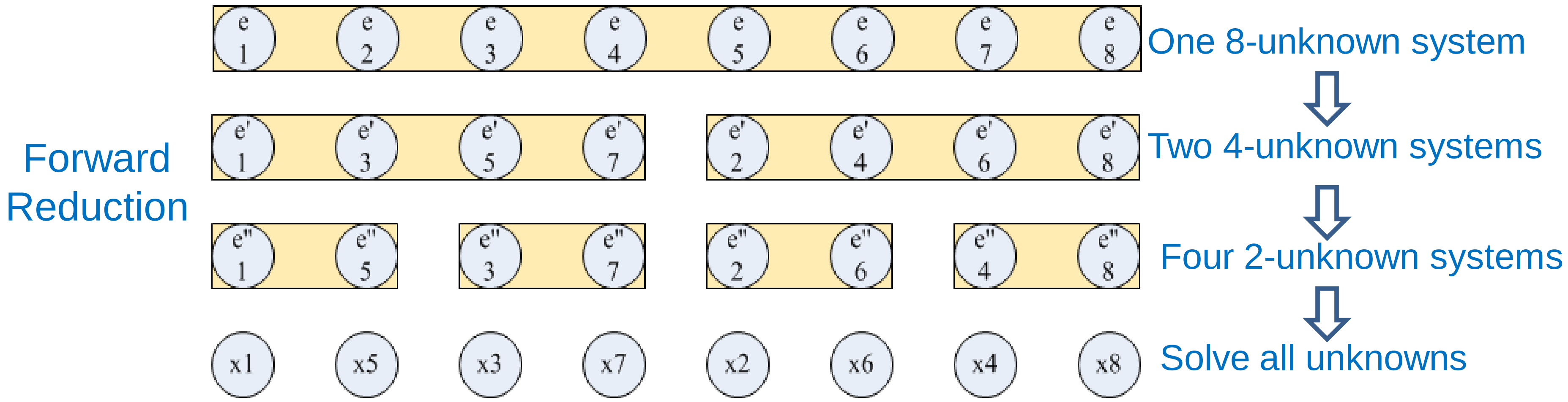
4 threads working



$$2*\log_2(8)-1 = 2*3-1 = 5 \text{ steps}$$

Parallel Cyclic Reduction (PCR)

4 threads working

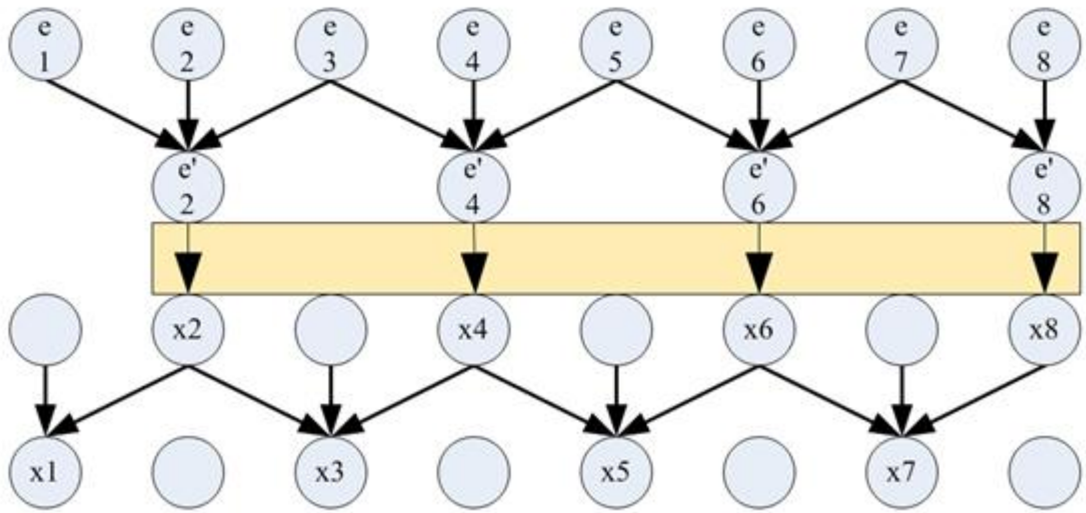


$\log_2(8) = 3$ steps

Hybrid Algorithm (1)

- CR
  - Every step we reduce the system size by half (Good)
  - Some processing cores stay **idle** if the system size is smaller than the number of cores (Bad)
  - Needs more steps to finish (Bad)
- PCR
  - Fewer steps required (Good)
  - Same amount of work for all steps (Bad)

Hybrid Algorithm (2)



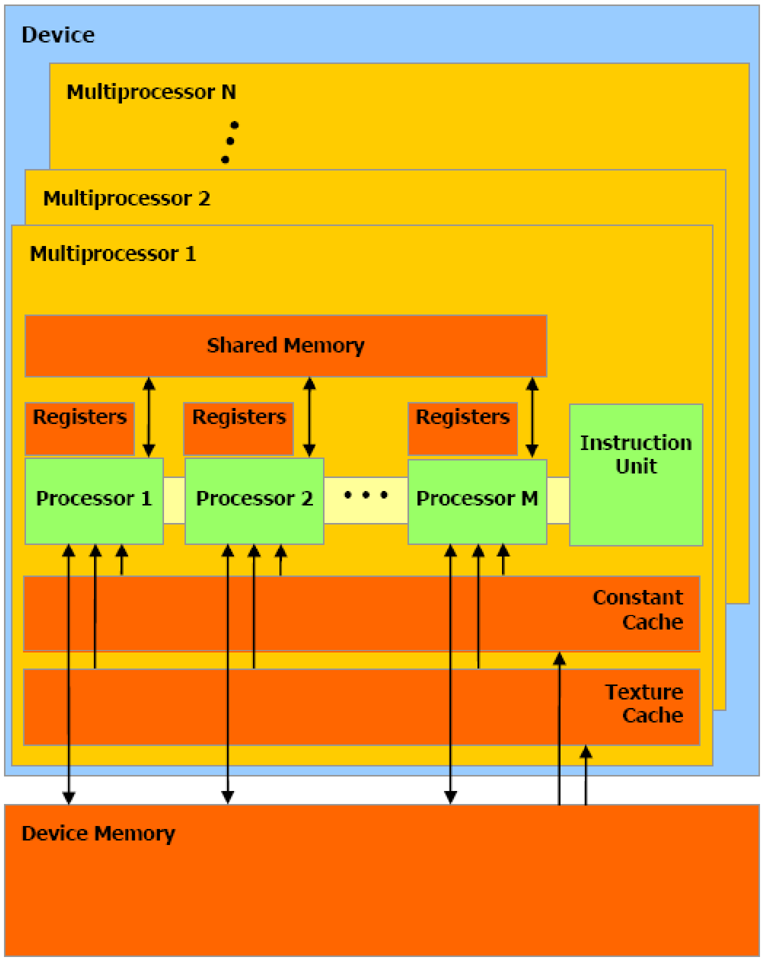
Switch to PCR

Switch back to CR

System size reduced at the beginning
No idle processors
Fewer algorithmic steps

Even more beneficial because of:
bank conflicts
control overhead

GPU Implementation (1)



- Linear systems mapped to multiprocessors (blocks)
- Equations mapped to processors (threads)

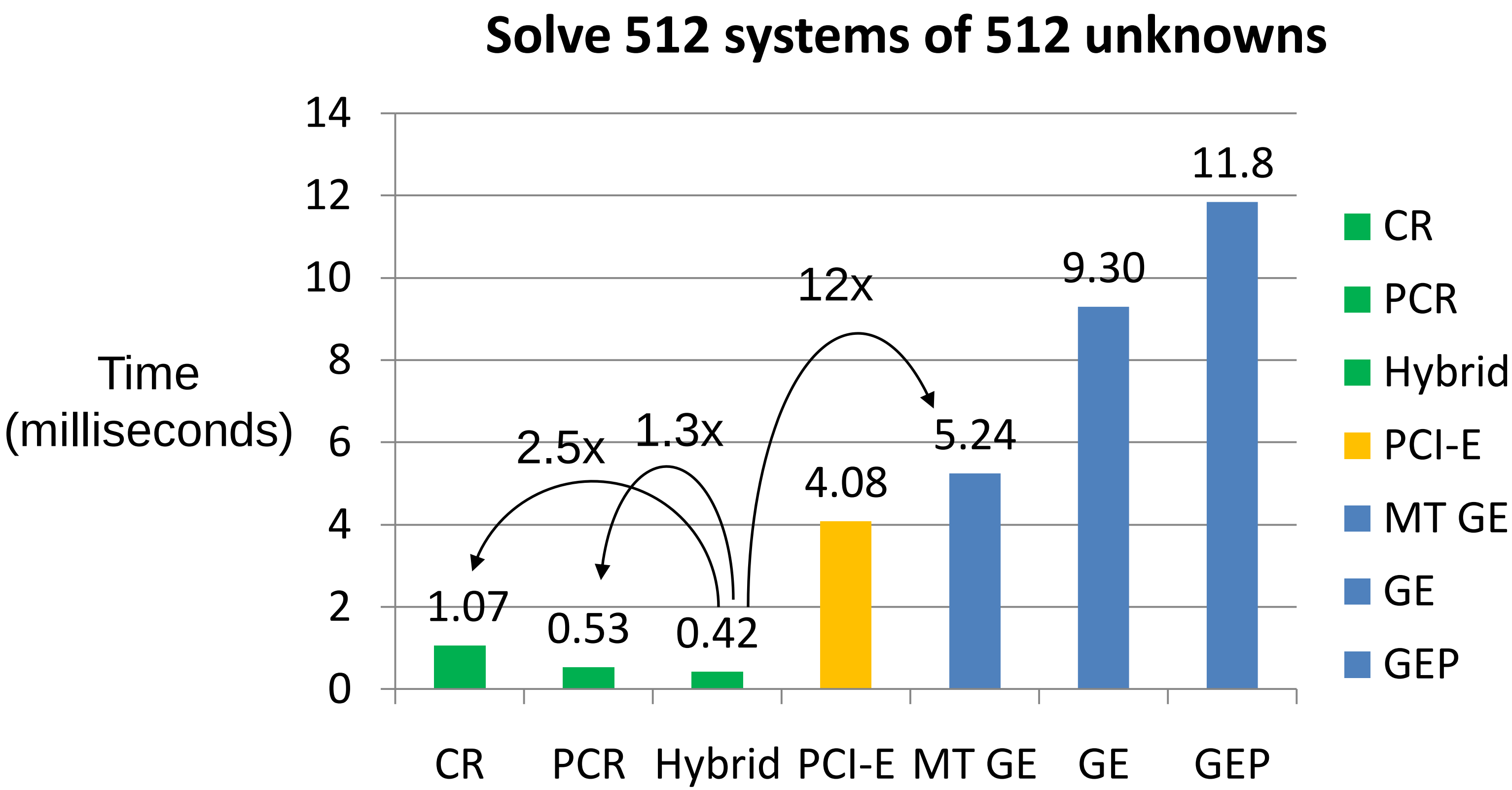
GPU Implementation (2)

- Storage need: 5 arrays = 3 diagonals + 1 solution vector + 1 right hand side
- All data resides in shared memory if it fits
- Use contiguously ordered threads to avoid unnecessary divergent branches
- In-place data storage
  - Efficient, but introduce bank conflicts to CR

Performance Results – Test Platform

- 2.5 GHz Intel Core 2 Q9300 quad-core CPU
- GTX 280 graphics card with 1 GB video memory
- CUDA 2.0
- CentOS 5 Linux operating system

Performance Results



PCI-E: CPU-GPU data transfer
MT GE: multi-threaded CPU Gaussian Elimination
GEP: CPU Gaussian Elimination with pivoting (from LAPACK)

Performance Analysis

- Factors that determine performance
  - Global/shared memory accesses
  - Bank conflicts
  - Computational complexity
  - Overhead for synchronization and loop control

Bottleneck vs. Pie slice

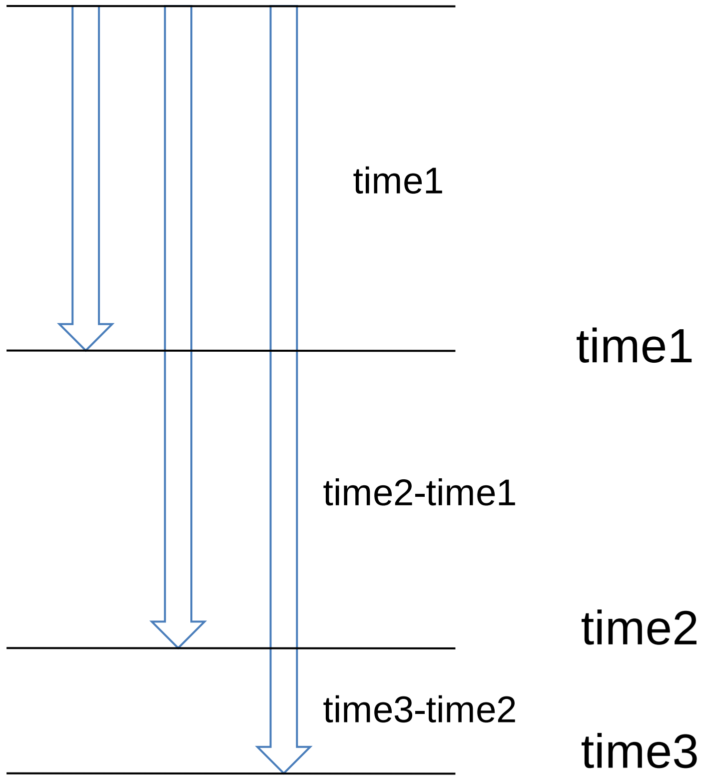
Performance = min(factor1, factor2, …)

Performance = sum(factor1, factor2, …)

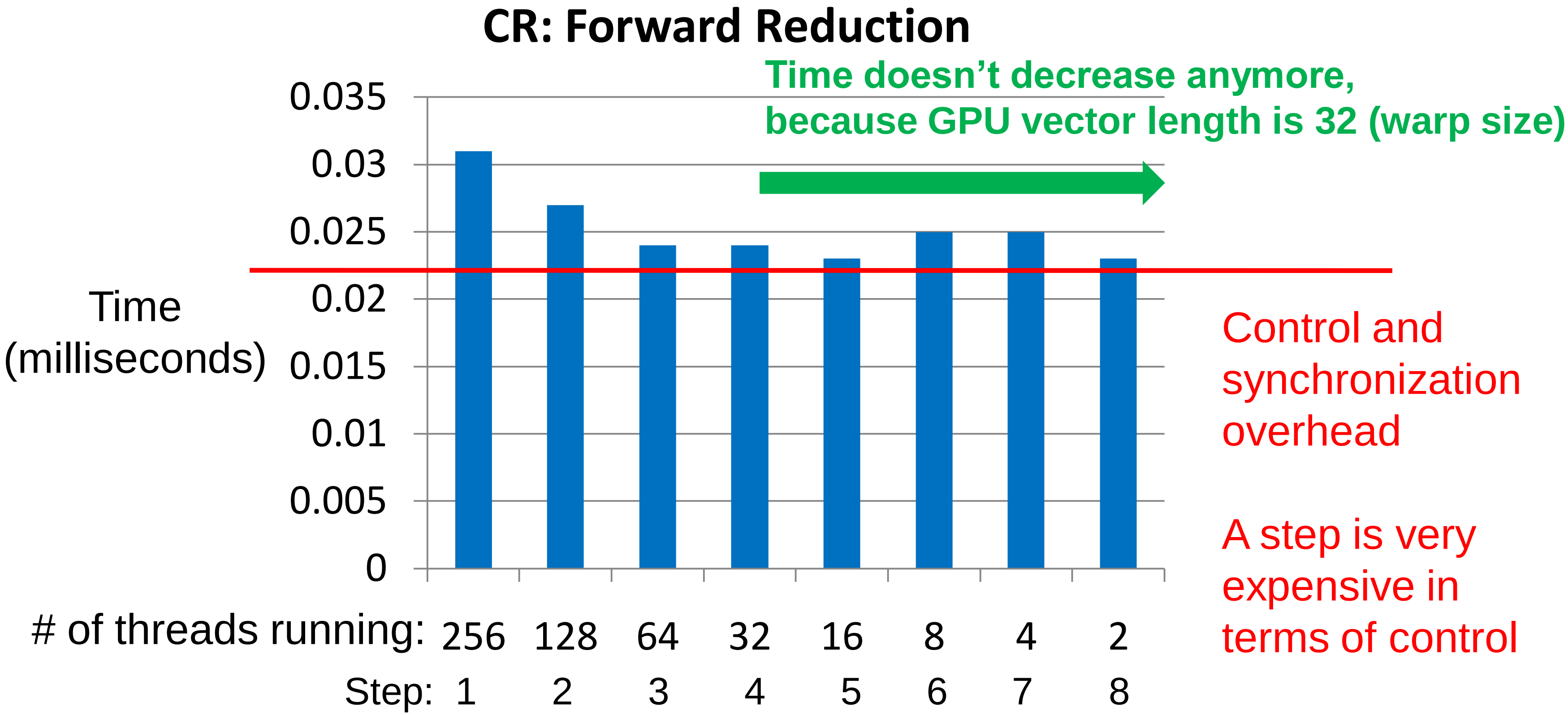
Performance Measure

A manual differential method

```
__global__ void scan_naive(float *g_odata, float *g_idata, int n)
{
    // Dynamically allocated shared memory for scan kernels
    extern  __shared__  float temp[];

    int thid = threadIdx.x;

    int pout = 0;
    int pin = 1;

    // Cache the computational window in shared memory
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;

    for (int offset = 1; offset < n; offset *= 2)
    {
        pout = 1 - pout;
        pin  = 1 - pout;
        __syncthreads();

        temp[pout*n+thid] = temp[pin*n+thid];

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
    }

    __syncthreads();

    g_odata[thid] = temp[pout*n+thid];
}
```

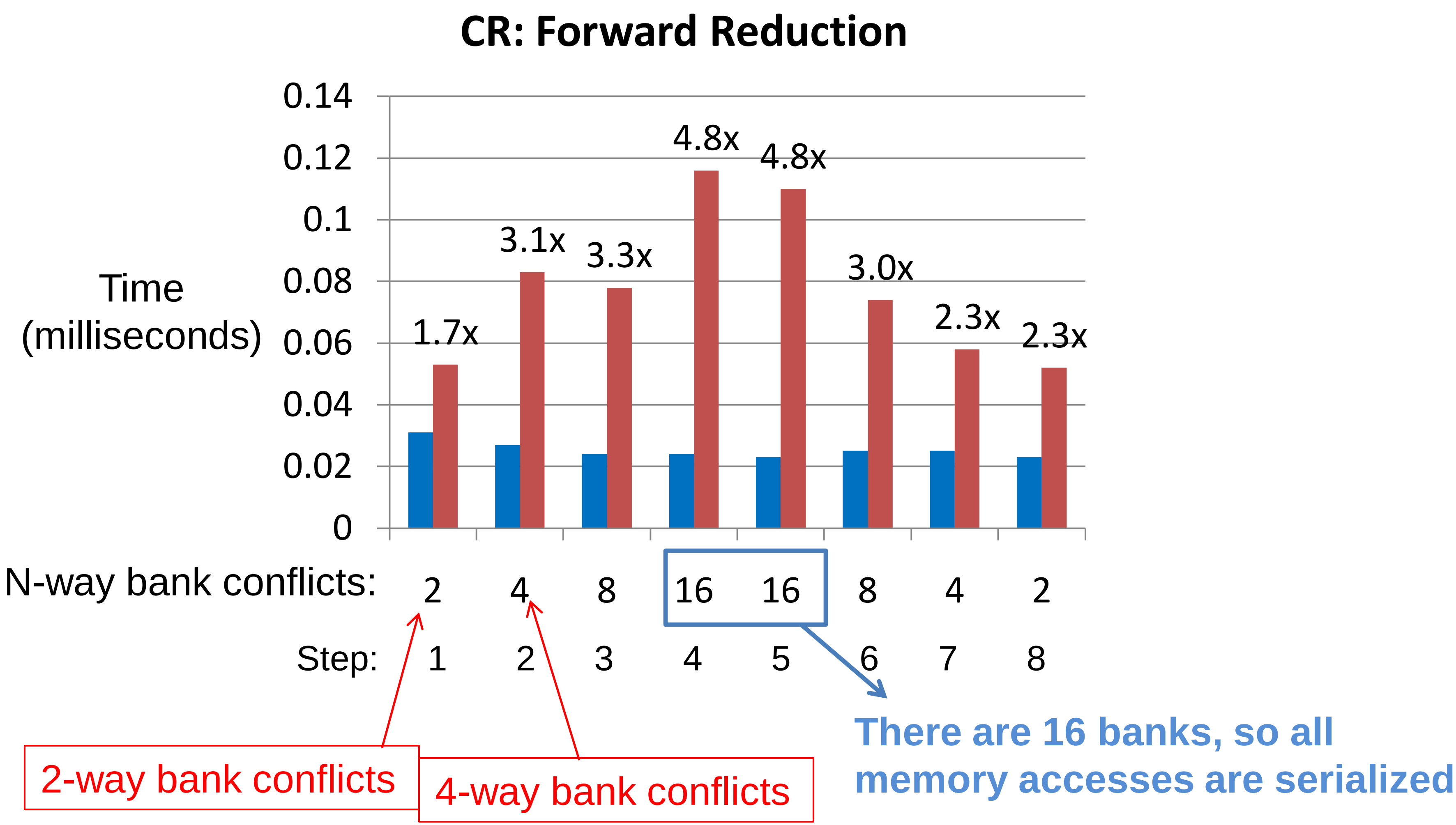


Control Overhead

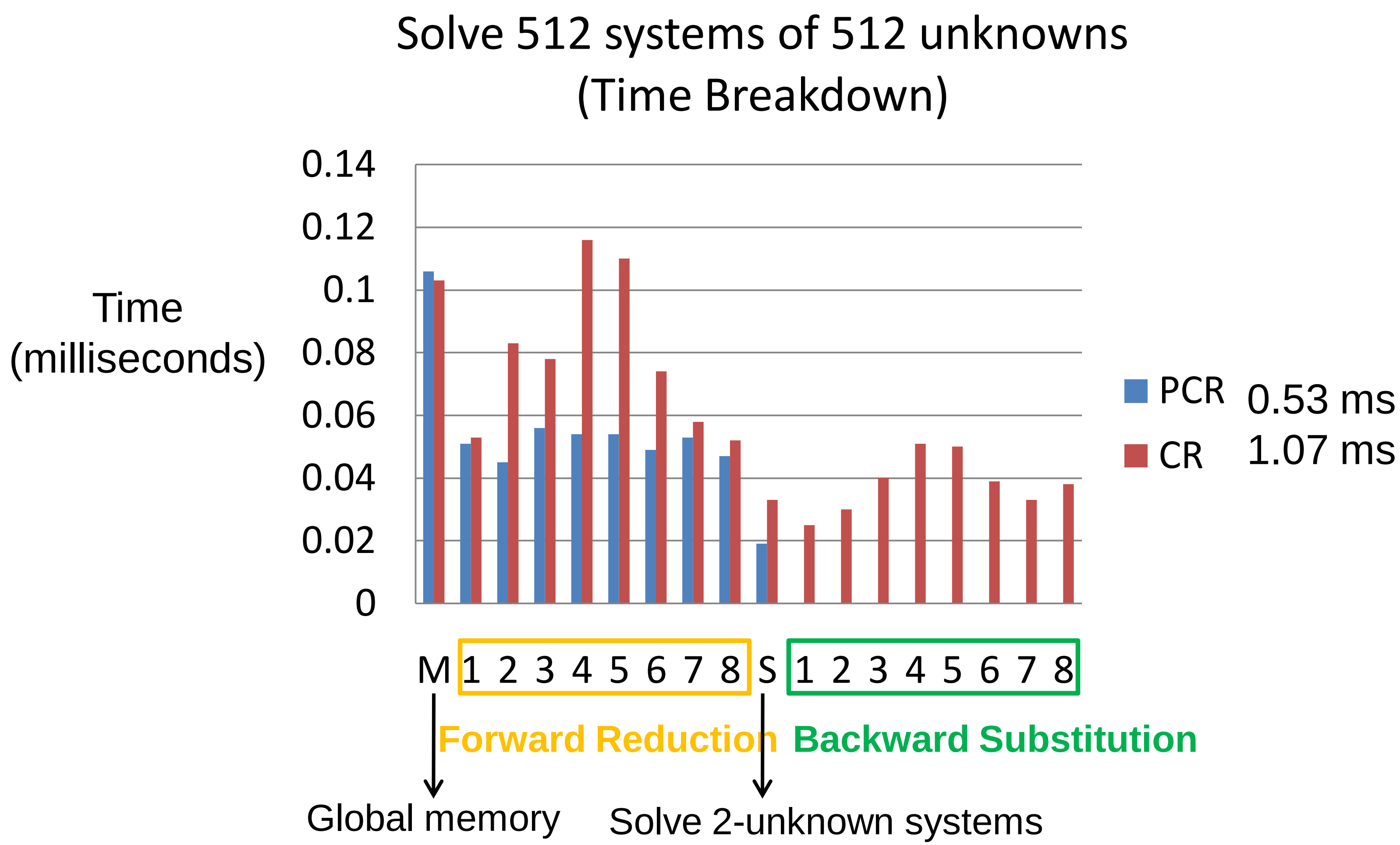


Enforce a stride of one to avoid bank conflicts

Bank Conflicts

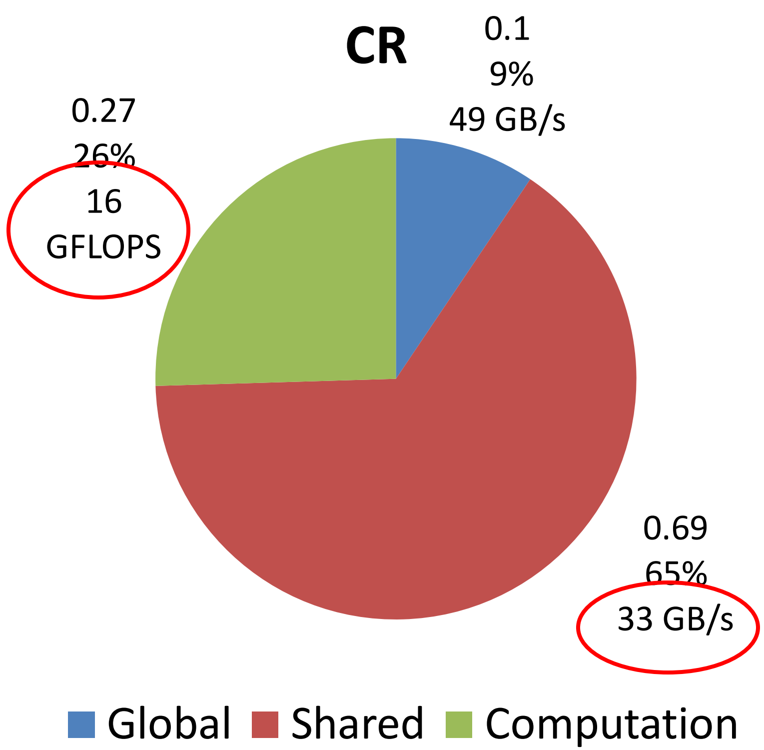


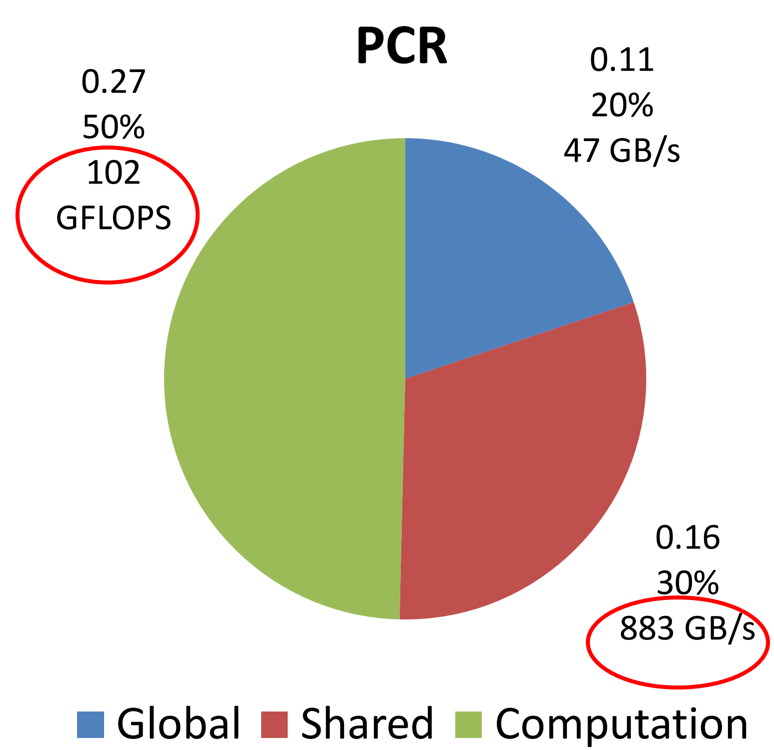
CR vs. PCR (1)



Solve 512 systems of 512 unknowns
(Time Breakdown)
Time
(milliseconds)
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
PCR
CR
0.53 ms
1.07 ms
M
1 2 3 4 5 6 7 8
S
1 2 3 4 5 6 7 8
Forward Reduction
Backward Substitution
Global memory
Solve 2-unknown systems

CR vs. PCR (2)





Pros: O(n)
Cons: more steps (control overhead), bank conflicts

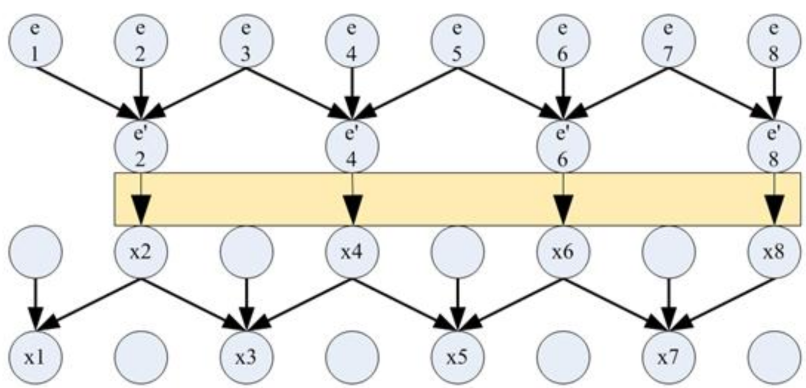
Pros: fewer steps, no bank conflicts
Cons: O(nlogn)

Pitfalls

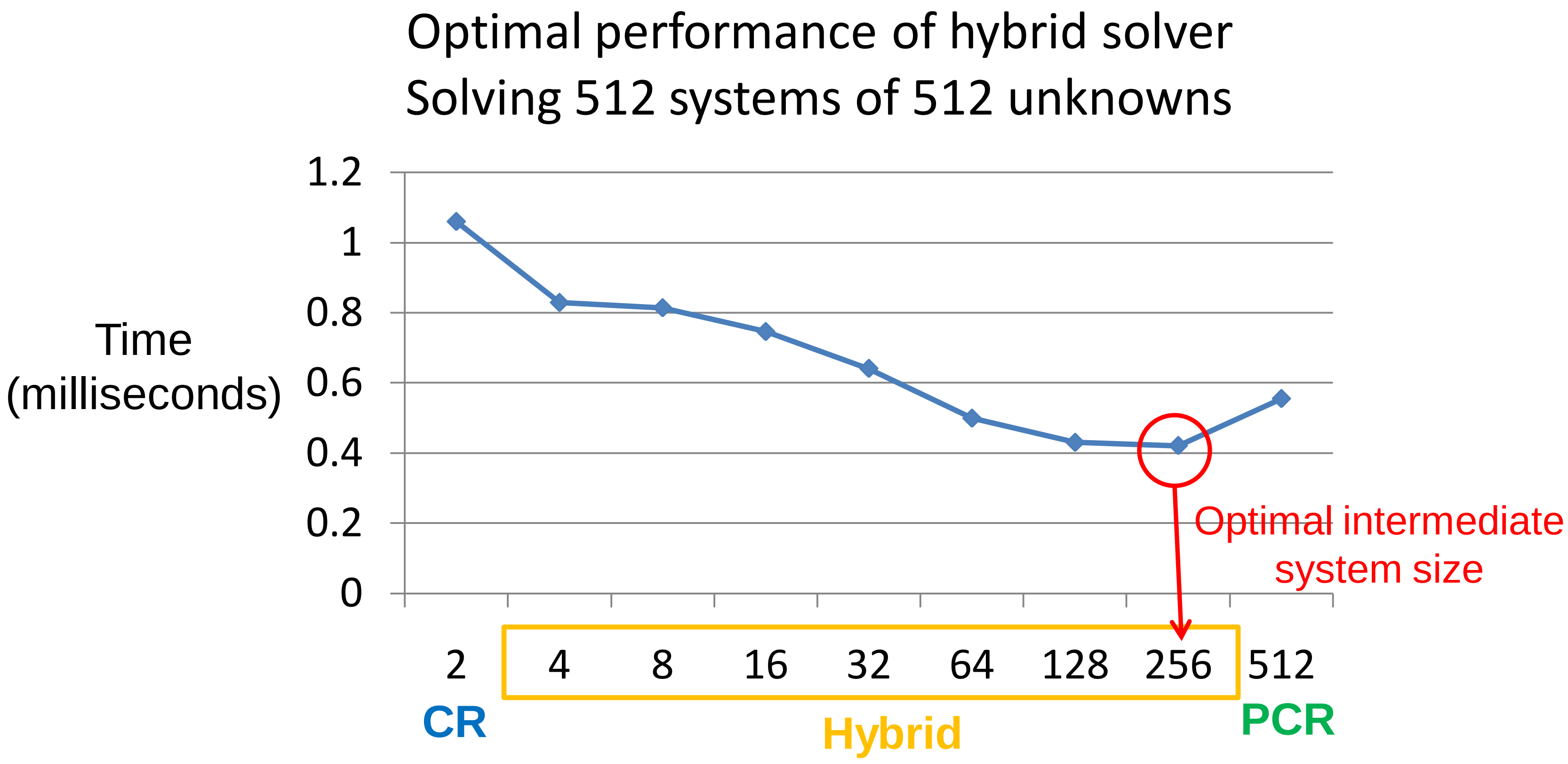
- The higher computation rate and sustained bandwidth, the better
  - They may have different algorithm complexity
- The lower algorithm complexity, the better
  - What if there is considerable amount of control overhead, or bank conflicts, or low hardware utilization

PCR vs Hybrid

- Make tradeoffs between the computation, memory access, and control
  - The earlier you switch from CR to PCR
    - The fewer bank conflicts, the fewer algorithmic steps
    - But more work



Hybrid Solver – Sweet Point



Known issues and future research

- PCI-E data transfer
- Double precision
- Pivoting
- Block tridiagonal systems
- Handle large systems that cannot fit into shared memory
- Automatic performance profiling

Summary

- We studied the performance of 3 parallel solvers on GPU
- We learned two major lessons
  - Component-based rather than bottleneck-based
    - » Performance is more complicated than either compute-bound or memory-bound
  - We can make tradeoffs between these components, and we need to make the right tradeoff

Questions?

Thanks